

SOFTWARE ARCHITECTURE THEMES IN JPL'S MISSION DATA SYSTEM

Daniel Dvorak, Robert Rasmussen, Glenn Reeves, Al Sacks
 Jet Propulsion Laboratory
 California Institute of Technology
 4800 Oak Grove Drive
 Pasadena, CA 91109-8099
 {dldvorak,rrasmssn,reeves,asacks}@pop.jpl.nasa.gov

ABSTRACT

The rising frequency of NASA mission launches has highlighted the need for improvements leading to faster delivery of mission software without sacrificing reliability. In April 1998 Jet Propulsion Laboratory initiated the Mission Data System (MDS) project to rethink the mission software lifecycle—from early mission design to mission operation—and make changes to improve software architecture and software development processes. As a result, MDS has defined a unified flight, ground, and test data system architecture for space missions based on object-oriented design, component architecture, and domain-specific frameworks. This paper describes several architectural themes shaping the MDS design and how they help meet objectives for faster, better, cheaper mission software.

BACKGROUND

JPL's deep space missions tend to be one-of-a-kind, each with distinct science objectives, instruments, and mission plans. Until recently, missions were spaced years apart, with little attention to software reuse, given the rapid pace of computer technology and computer science. Also, since radiation-hardened flight computers remain years behind their commercial counterparts in speed and memory, flight software has typically been highly customized and tuned for each mission. Thus, when JPL launched six missions in six months between October 1998 and March 1999, it wasn't surprising that there was little software reuse among them, except in the ground system.

However, despite the uniqueness of each mission, they each had to independently design and develop mechanisms for communication, commanding,

attitude control, navigation, power management, fault protection, and many other standard tasks, yet there was no common architecture or frameworks for them to draw upon. Clearly, in an era of monthly missions, this is an inefficient way to use software-engineering resources.

MISSION DATA SYSTEM

In order to use software-engineering resources more effectively and to sustain a quickened pace of missions, JPL initiated a project in April 1998 to define and develop an advanced multi-mission architecture for an end-to-end information system for deep-space missions. The system, named "Mission Data System" (MDS), is aimed at several institutional objectives: earlier collaboration of mission, system and software design; simpler, lower cost design, test, and operation; customer-controlled complexity; and evolvability to in situ exploration and other autonomous applications. JPL's Telecommunication and Mission Operations Directorate (TMOD) manages the MDS project.

This paper describes several architectural themes shaping the MDS design and how they help meet these objectives and how they differ from earlier practices. Although most of these themes have resulted from a desire to improve flight software—and have compelling examples there—they apply equally to ground software. Also, these themes apply equally to all kinds of robots, whether spacecraft or probes or rovers.

AN ARCHITECTURAL APPROACH

Theme: Construct subsystems from architectural elements, not the other way around.

It has been traditional in JPL missions to divide the work along five dimensions: flight vs. ground vs. test, design vs. test vs. operations, engineering vs. science, downlink vs. uplink, and subsystems (navigation vs. power vs. propulsion vs. telecom, etc). With the work so compartmentalized, software engineers naturally

Copyright © 1999 by the American Institute of Aeronautics and Astronautics, Inc. The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental purposes. All other rights are reserved by the copyright owner.

applied their own customized solutions within each realm, resulting in minimal reuse and requiring many iterations at integrating the subsystems. The net result was always architecture constructed from subsystems.

In MDS there is a quest to find common problems and create common solutions, and also to tailor general solutions to particular problems. This quest is driven by the recognition that managing interactions is the foundation of good design. For example, different activities in different subsystems issue commands that consume power, and they can potentially interfere with each other unless there is a coordination service that keeps track of available power and who has authority to control each device. Creating such a coordination service enables a cleaner simpler design because it controls interactions through a common service rather than through private subsystem-to-subsystem agreements, thereby decreasing the coupling between subsystems. It similarly simplifies unit testing of subsystems. The net result from applying this approach is that subsystems get constructed from architectural elements, not the other way around.

GROUND-TO-FLIGHT MIGRATION

Theme: Migrate capability from ground to flight, when appropriate, to simplify operations.

MDS takes a unified view of flight and ground tasks because of opportunity and need. With increasingly powerful flight processors the *opportunity* exists to migrate to the spacecraft (or rover) some processing that has traditionally been performed on the ground, thereby reducing the need for flight-ground communication. Such migration might occur well after launch, after ground operators have gained experience with the real spacecraft and have decided that some activities can be automated, without further human-in-the-loop control. More importantly, the *need* for such migration exists in order to accomplish missions that must react quickly to events, without earth-in-the-loop light-time delays, such as autonomous landing on a comet and rover explorations on Mars. For these reasons both flight and ground capabilities must be designed for a shared architecture.

STATE & MODELS ARE CENTRAL

Theme: System state and models form the foundation for monitoring and control.

MDS is founded upon a state-based architecture, where *state* is a representation of the momentary condition of an evolving system. System states include device operating modes, device health states, resource levels, temperatures, pressures, etc, as well as

environmental states such as the motions of celestial bodies and solar flux. Some aspects of system state are best described as functions of other states; e.g., pointing can be derived from attitude and trajectory. In all cases state is accessible through *state variables* (as opposed to a program's local variables), and state evolution is described on *state timelines*. State timelines provide the fundamental coordinating mechanism since they describe both knowledge and intent.

A state-based architecture implies the need for models since models describe how a system's state evolves. Together, state and models supply what is needed to operate a system, predict future state, control toward a desired state, and assess performance.

EXPLICIT USE OF MODELS

Theme: Express domain knowledge explicitly in models rather than implicitly in program logic.

Much of what makes software different from mission to mission is domain knowledge about instruments and actuators and sensors and plumbing and wiring and many other things. This knowledge includes relationships such as how power varies with solar incidence angle, conditions such as the fact that gyros saturate above a certain rate, state machines that prescribe safe sequences for valve operation, and dynamic models that predict how long a turn will take. Conventional practice has been to develop programs whose logic implicitly contains such domain knowledge, but this expresses the knowledge in a "hidden" form that is hard to validate and hard to reuse.

In contrast, MDS advocates that domain knowledge be represented more explicitly in inspectable models. Such models can be tables or spreadsheets or rules or state machines or any of several forms, as long as they separate the domain knowledge from the general logic for applying that knowledge to solve a problem. The task of customizing MDS for a mission, then, becomes largely a task of defining and validating models.

GOAL-DIRECTED OPERATION

Theme: Operate missions via specifications of desired state rather than sequences of actions.

Traditionally, spacecraft have been controlled through linear (non-branching) command sequences that have been carefully designed on the ground. Such design is difficult for two reasons. First, ground must predict spacecraft state for the time at which the sequence is scheduled to start, and that's difficult to know with confidence because of flight/ground communication

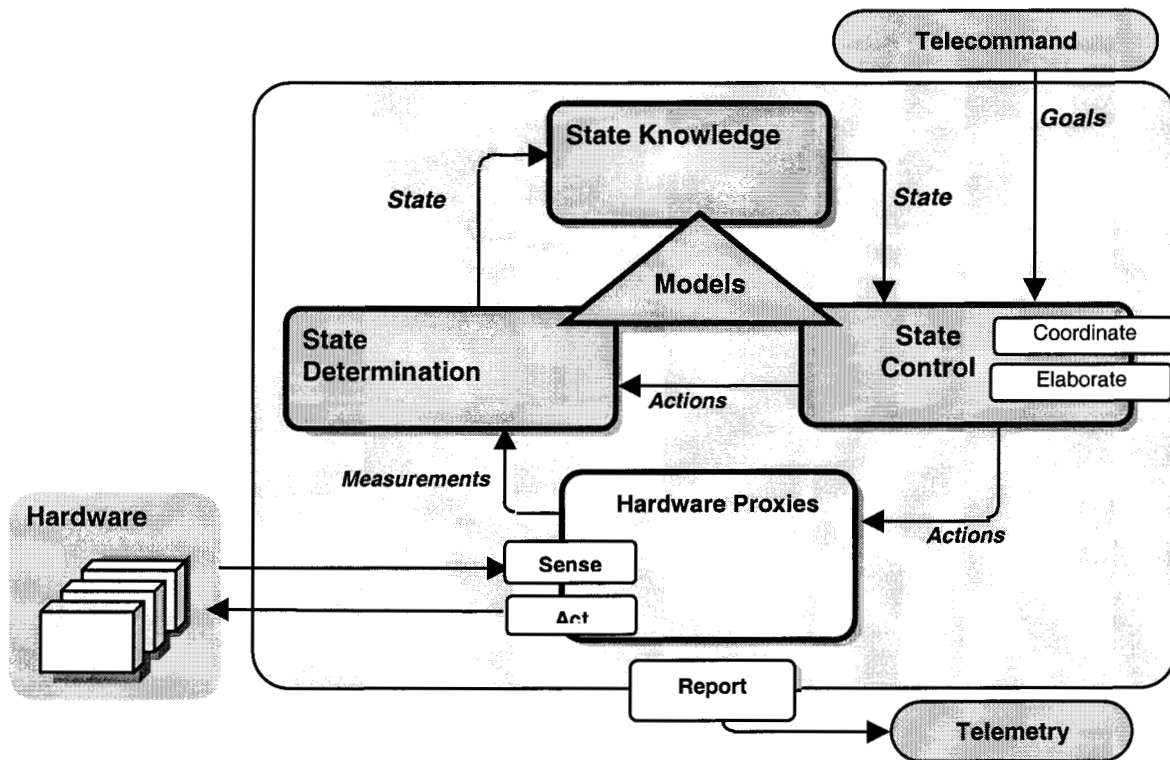


Figure 1. This figure illustrates several MDS architectural themes: the central role of state knowledge and models, goal-directed operation, closed-loop control, and the separation of state determination from state control.

limitations (data rate and light-time delay). Second, in the event that the actual spacecraft state is different than the predicted state, the sequence should be designed to fail rather than chance doing something harmful.

MDS, in contrast, controls state—both flight and ground state—via “goals”. A *goal* is defined as a prioritized constraint on the value of a state variable during a time interval. A goal differs from a command in that it specifies *intent* in the form of desired state. Such goal-directed operation is simpler than traditional sequencing because a goal is easier to specify than the actions needed to accomplish it. Importantly, goals specify only success criteria; they leave options open about the means of accomplishing the goal and the possible use of alternate actions to recover from problems. A goal is a unifying concept that encompasses daily operations, maintenance and calibration, resource allocation, flight rules, and fault responses. Of course, all of this begs the question of who or what elaborates a goal into a program of actions, which brings us to closed-loop control and goal-achieving modules.

CLOSED-LOOP CONTROL

Theme: Design for real-time reaction to changes in state rather than for open-loop commands or earth-in-the-loop control.

Goal-directed operation implies closed-loop control. In MDS a state controller is termed a *goal-achieving module* (GAM). A GAM controls state by comparing present state to desired state, then deciding how to change the state if necessary, then issuing either sub-goals to lower-level GAMs or issuing direct low-level actions (i.e., primitive actions). When a GAM accepts a goal it must either achieve the goal or responsibly report that it cannot. A GAM’s logic can be arbitrarily simple or sophisticated, but it must always keep the goal issuer informed about the goal’s status.

Most GAMs achieve their goals by issuing sub-goals, thus creating a hierarchy of GAMs. Naturally, the bottom layer of GAMs bottoms out in primitive actions. Importantly, GAMs can report why they acted as they did in terms of what discrepancies between state and goals prompted action, and what sub-goals or commands were issued in response. Also, since GAMs

are self-checking by definition, goal failures will be visible (through goal status) during testing.

INTEGRAL FAULT PROTECTION

Theme: Fault protection must be an integral part of the design, not an add-on.

Fault protection, which includes fault detection, localization, and recovery, has often been treated as an add-on to a basic command & control system. As such, it was designed *after* the control system and arrived later in the project cycle. Such was the case for the Cassini attitude and articulation control system, and an interesting thing happened the day that fault protection was first enabled: numerous faults were detected in a control system that had already undergone a fair amount of testing. The Cassini AACS team learned more in that month than they had in the previous six months because they finally had independent detailed monitoring of system behavior.

In MDS fault protection will be an integral part of the design—not an add-on—because it is an essential part of robust control and because it is extremely valuable during system testing. Goal-achieving modules in MDS need at least some minimum level of fault detection since they *must* report when an active goal is not being achieved. GAMs may also provide recovery strategies ranging from very simple to very sophisticated.

REAL-TIME RESOURCE MANAGEMENT

Theme: Resource usage must be authorized and monitored by a resource manager.

“Resources” are things like available battery energy, power, fuel, memory, thermal margin, etc. Overuse of spacecraft resources can be disastrous, such as accidentally using too much power near the time of a critical orbit insertion maneuver, causing the spacecraft power bus to trip. For reasons like this ground operators have tended to be very conservative about resource usage, especially given their time-delayed knowledge of it. However, such conservative operation limits the amount of science data acquisition and return, especially during periods of great opportunity, such as during a fly-by or a short-lived science event.

MDS avoids this kind of operational dilemma through a resource management mechanism that prevents overuse, even if a resource is accidentally oversubscribed. Specifically, resource-using activities are forced obtain a “ticket” in order to use a given

resource, much as one obtains a file descriptor in order to access a file. An activity must state to a resource manager the amount of resource and the time interval when it is needed, and the ticket is issued only if the usage does not conflict with any other higher-priority usage. Further, if measurements show that more of a resource is being used than was ticketed (such as might occur from an unexplained power drain), the manager can disable one or more tickets until an adequate margin is recovered. Because a resource manager always knows the available amount, other activities can be triggered to opportunistically use the resource, thereby increasing science data return.

SEPARATION OF STATE DETERMINATION AND CONTROL

Theme: For consistency, simplicity and clarity, separate state determination logic from control logic.

It’s not unusual to see software that co-mingles control logic with state determination logic, but this practice is usually a bad idea for three reasons. First, if two or more controllers each make their own private determination for the same state variable, their estimates may differ, potentially leading to conflicting control actions. Second, mixing two distinct tasks in the same module makes the code harder to understand and less reusable. Third, these two tasks are an ill fit in the same module because control has a hierarchical structure based on delegation of authority whereas state determination has a network structure based on pathways of interaction mechanisms (electrical, thermal, etc.).

Architecturally, MDS separates state determination from state control, coupled only through state variables. State determination is a process of interpreting measurements to generate state knowledge, and the process may combine multiple sources of evidence into a determination of state, supplied to a state variable as an estimate. Control, in contrast, attempts to achieve goals by issuing commands and sub-goals that should drive estimated state toward desired state. Keeping these two tasks separate simplifies design, programming, and testing, and also allows for independent improvements.

ACKNOWLEDGE STATE UNCERTAINTY

Theme: State determination must be honest about the evidence; state estimates are not facts.

State values are rarely known with certainty, but a lot of software effectively pretends that they are by

treating state estimates as facts. However, disastrous errors can result when control decisions are based on highly uncertain state. For example, it is probably unwise to perform a main-engine burn when the estimated position of the engine gimbals is below some minimum certainty. Uncertainty can arise in several ways, sometimes as conflicting evidence, sometimes through characteristic degradation of sensors, and sometimes during periods of rapid dynamic change.

MDS takes the position that a level of certainty should accompany every state estimate. State determination must be honest about what the evidence is telling it. If there are two credible pieces of evidence that conflict, and there's no timely way to reconcile the conflict, then the resulting state estimate must have an appropriately reduced level of certainty. Similarly, control must take into account the certainty level of the state estimates upon which it is basing a decision. If certainty drops below some context-specific minimum, then control must react appropriately, perhaps by attempting an alternate approach or by abandoning a goal entirely.

SEPARATION OF DATA MANAGEMENT FROM DATA TRANSPORT

Theme: Separate data management duties and structures from those of data transport.

Flight/ground data management has long been tightly coupled with data transport issues, largely because such capabilities evolved from a time when flight processors were extremely limited. This resulted in application code that was built around the CCSDS packet format, for example. While such designs had some justification in the speed and memory constraints of earlier missions, the time has come to adopt a cleaner separation and prepare for the day when spacecraft are in fact nodes in an inter-planetary network.

MDS clearly distinguishes between *data management* and *data transport*. The former elevates data products as entities in their own right, as objects and files that can be updated and summarized and aged, and that may or may not be destined for ground. In fact, data management is a service that transcends the flight-ground divide so that data products are treated consistently in both places. Data transport, in contrast, can access any data product and serialize it for transport between flight and ground. Packet formats and link protocols are completely hidden from the level of data management. Decoupling these two

capabilities keeps the design and testing simpler for each and allows for independent improvements.

JOINING NAVIGATION WITH ATTITUDE CONTROL

Theme: Navigation and attitude control must build from a common mathematical base.

Navigation and attitude control have been weakly coupled on most JPL missions because, in empty space, they operate on vastly different time scales and their dynamics don't greatly affect each other. As such, navigation software and attitude control software have been developed largely independent development efforts. In upcoming missions, however, the coupling becomes much tighter. For example, escape velocity near an asteroid is so small that firing thrusters for attitude control can significantly affect trajectory. Likewise, docking with another vehicle, as in a sample-return mission, requires navigation and attitude corrections on similar time scales.

The approach that MDS is taking here, as in other areas, is to design common architectural mechanisms for common problems. Since the same forces influence navigation and attitude control, the architecture must allow for a common model; since both are solving geometry problems, the architecture must provide for common solvers.

UPWARD COMPATIBILITY

Theme: Design interfaces to accommodate foreseeable advances in technology.

MDS is intended to serve missions for many years to come, and during that time there will be numerous advances in software technology for control systems, fault detection & diagnosis, planning & scheduling, databases, communication protocols, etc. MDS must be prepared to exploit such technologies else it will become an obstacle rather than an enabler for increasingly challenging missions, but MDS also needs to maintain some architectural stability to amortize its cost over its missions. The strategy for achieving this centers around careful design of architectural interfaces, behind which a variety of technical approaches can be used. Specifically, MDS designers consult with researchers to understand how software interfaces may need to evolve, and then implement a restrict subset of an interface using current mission-ready technology. When the more advanced technology becomes mission-ready, they implement the fuller interface in an upward

compatible manner, namely, in a manner that still works for interface clients that use the restricted subset. Thus, interface client software is not forced to change on the same schedule as interface provider software.

The value of upward compatibility is powerfully illustrated in the history of IBM. In 1964 when IBM introduced the System/360 architecture, they transformed the computer industry with the first "compatible" family of computers. Software and peripherals worked virtually interchangeably on any of the five original processors, so customer investments were preserved when they upgraded to a more powerful processor. IBM continued to improve the technology over the years, but always within the System/360 architecture and its extensions. Although the MDS architecture applies to a much smaller marketplace, the benefits of upward compatibility make sense for MDS customers *and* providers.

BENEFIT TO MDS CUSTOMERS

The main value of MDS is that it should enable customer missions to focus on mission-specific design and development without having to create and test a supporting infrastructure. Customers will receive a set of pre-integrated and pre-tested frameworks, complete with executable example uses of those frameworks running on a simulated spacecraft and mission. These frameworks will be based on an object-oriented design described in Unified Modeling Language (UML) [OMG, 1997], the *lingua franca* of MDS software design and scenario description.

As a project, MDS is balancing a long-term architectural vision against a near-term commitment to its first customer mission, Europa Orbiter, scheduled to launch in 2003. Such commitments help focus MDS design efforts on pragmatic, well-understood mechanisms for supporting the architectural themes.

HISTORICAL CONTEXT

In a 1995 joint study between NASA Ames and JPL known as the New Millennium Autonomy Architecture Prototype (NewMAAP) a number of existing concepts for improving flight software were brought together in a prototype form. These concepts included goal-based commanding, closed-loop control, model-based diagnosis, onboard resource management, and onboard planning. When the Deep Space One (DS-1) mission was subsequently announced as a technology validation mission, the

NewMAAP project rapidly segued into the Remote Agent project [Pell et al, 1997]. In May 1999 the Remote Agent eXperiment (RAX) flew on DS-1 and provided the first in-flight demonstration of the concepts. The MDS project, which is populated with many people who worked on or with RAX, was established in April 1998 to define and develop an advanced multi-mission data system that unifies the flight, ground, and test elements in a common architecture. That architecture is shaped with the themes described in this paper.

ACKNOWLEDGEMENTS

The research and design described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

[OMG, 1997] "What is OMG-UML and why is it important?", Object Management Group, <http://www.omg.org/news/pr97/umlprimer.html>, 1997.

[Pell et al, 1997] "An Autonomous Spacecraft Agent Prototype," B. Pell, D. Bernard, S. Chien, E. Gat, N. Muscettola, P. Nayak, M. Wagner, B. Williams, *Proceedings of the First Annual Workshop on Intelligent Agents*, Marina Del Rey, CA, 1997.